Provably Efficient Scheduling of Cache-oblivious Wavefront Algorithms

Rezaul Chowdhury
Stony Brook University, Stony Brook, NY 11790
rezaul@cs.stonybrook.edu

Pramod Ganapathi
Stony Brook University, Stony Brook, NY 11790
pganapathi@cs.stonybrook.edu

Yuan Tang
School of Comp. Science, School of Software, Fudan Univ.
Shanghai Key Lab. of Intelligent Information Processing
State Key Lab. of Comp. Arch. ICT, CAS, China
yuantang@fudan.edu.cn

Jesmin Jahan Tithi
Intel Corporation
California, USA
jesmin.jahan.tithi@intel.com

ABSTRACT

Iterative wavefront algorithms for evaluating dynamic programming recurrences exploit optimal parallelism but show poor cache performance. Tiled-iterative wavefront algorithms achieve optimal cache complexity and high parallelism but are cache-aware and hence are not portable and not cache-adaptive. On the other hand, standard cache-oblivious recursive divide-and-conquer algorithms have optimal serial cache complexity but often have low parallelism due to artificial dependencies among subtasks.

Recently, we introduced cache-oblivious recursive wavefront (COW) algorithms, which do not have any artificial dependencies, but they are too complicated to develop, analyze, implement, and generalize. Though COW algorithms are based on fork-join primitives, they extensively use atomic operations for ensuring correctness, and as a result, performance guarantees (i.e., parallel running time and parallel cache complexity) provided by state-of-the-art schedulers (e.g., the randomized work-stealing scheduler) for programs with fork-join primitives do not apply. Also, extensive use of atomic locks may result in high overhead in implementation.

In this paper, we show how to systematically transform standard cache-oblivious recursive divide-and-conquer algorithms into recursive wavefront algorithms to achieve optimal parallel cache complexity and high parallelism under state-of-the-art schedulers for fork-join programs. Unlike COW algorithms these new algorithms do not use atomic operations. Instead, they use closed-form formulas to compute the time when each divide-and-conquer function must be launched in order to achieve high parallelism without losing cache performance. The resulting implementations are arguably much simpler than implementations of known COW algorithms. We present theoretical analyses and experimental performance and scalability results showing a superiority of these new algorithms over existing algorithms.

Keywords: wavefront; cache-oblivious; parallel; recursive; divide-and-conquer; dynamic programming; parallelism

SPAA '17, July 24-26, 2017, Washington DC, USA

DOI: 10.1145/3087556.3087586

1 INTRODUCTION

Dynamic programming (DP) is a popular algorithm design technique to solve optimization problems that exhibit the properties of overlapping subproblems and optimal substructure. The process involves dividing a problem into smaller subproblems, solving them, storing their results in tables to avoid recomputations, and combining those solutions. DP is used in many real-world application areas, and extensively in computational biology [6, 22, 31, 52].

For good performance on a modern multicore machine with a cache hierarchy, algorithms must have good parallelism and should be able to use the caches efficiently at the same time. Iterative wavefront algorithms for solving DP problems have optimal parallelism but often suffer due to poor cache performance. On the other hand, though standard cache-oblivious [24] recursive divide-and-conquer DP algorithms have optimal serial cache complexity, they often have low parallelism. The tiled-iterative wavefront algorithms achieve optimality in cache complexity and achieve high parallelism but are cache-aware, and hence are not portable and do not adapt well when available cache space fluctuates during execution in a multiprogramming environment. Very recently, the *cache-oblivious wavefront (COW) algorithms* [46, 47] have been proposed that have optimal parallelism and optimal serial cache complexity. However, though those algorithms are based on fork-join primitives, they extensively use atomic operations for correctness. But, the current theory of scheduling nested parallel programs with fork-join primitives does not hold for such atomic operations. As a result, no bounds on parallel running time and parallel cache complexity could be proved for those algorithms. Those algorithms are also very difficult to implement since they require hacking into a parallel runtime system. Extensive use of atomic locks causes too much overhead for very large and higher dimensional DPs.

We present a provably efficient method for scheduling cache-oblivious recursive divide-and-conquer wavefront algorithms on a multicore machine which optimizes parallel cache complexity and achieves high parallelism. Our algorithms are based on fork-join primitives but do not use atomic operations. As a result, we are able to analyze their parallel running times and parallel cache complexities easily under the state-of-the-art schedulers for fork-join based parallel programs. Our algorithms are also much simpler to implement compared to COW algorithms.

Performance of a parallel program on multicores. We analyze the performance of a parallel program run on a shared-memory multicore machine using the *work-span* model [19]. The *work* of

a multithreaded program, denoted by $T_1(n)$, where n is the input parameter, is the total number of CPU operations performed when run on a single processor[1]. The *span* (a.k.a. critical-path length or depth), denoted by $T_\infty(n)$, is the maximum number of operations performed on any processor when the program is run on an infinite number of processors. The *parallel running time* $T_p(n)$ of a program when scheduled by a greedy scheduler [11] on p processors is given by $T_p(n) = O(T_1(n)/p + T_\infty(n))$. The *parallelism*, computed by the ratio of $T_1(n)$ and $T_\infty(n)$, is defined as the average amount of work done per step of the critical path. The notations and their meanings are summarized in Table 1.

Cache complexity is a performance metric that counts the number of block transfers (or cache misses or I/O transfers or page faults) triggered by a program between adjacent levels of caches in a memory hierarchy. By Q_p we denote the total number of cache misses on a p-processor machine. So Q_1 is the *serial cache complexity*. We say that an algorithm has *spatial locality* provided each cache block it brings into a cache contains as much useful data as possible. We say that it has *temporal locality* provided it performs as much useful work as possible on each cache block it brings into a cache before the block gets evicted from the cache.

Symb.	Meaning
n	Input parameter
n'	Switching point (to non-wavefront)
r	Parameter $\in [1, n]$
p	Number of processors
M	Cache size
B	Cache line size
T_1	Work or total #computations
T_∞	Span or critical-path length
T_p	Parallel running time
T_1/T_∞	Parallelism
Q_1	Serial cache complexity
Q_p	Parallel cache complexity
S_p	Extra-space complexity

Table 1: Standard notations used throughout the paper.

Iterative algorithms. Traditionally, DP algorithms are implemented using a series of (nested) loops and they can be parallelized easily. These algorithms often have good spatial locality, but no temporal locality and standard implementations may not have optimal parallelism either. For example, an iterative algorithm for the parenthesis problem [49] (shown in Figure 1 and explained in Section 2) has $T_\infty(n) = \Theta\left(n^2\right)$ and $Q_1(n) = \Theta\left(n^3\right)$.

Iterative algorithms are also implemented as tiled loops, in which case the entire DP table is blocked or tiled and the tiles are executed iteratively. For example, for a tiled iterative algorithm for the parenthesis problem with $r \times r$ tile size, where $r \in [2, n]$, we have $T_\infty(n) = \Theta\left((n/r)^2\right) \cdot \Theta\left(r^2\right) = \Theta\left(n^2\right)$, and $Q_1(n, r) = (n/r)^3 \cdot O\left(r^2/B + r\right) = O\left(n^3/(rB) + n^3/r^2\right)$.

Fastest iterative DP implementations have the following wavefront-like property. Let a single update on a cell x in the DP table needs to be applied by reading from cells $y_1, y_2, \ldots, y_s$. When the cells $y_1, y_2, \ldots, y_s$ are completely updated, then the cell x can immediately get updated, either partially or fully.

Recursive algorithms. Cache-oblivious parallel recursive divide-and-conquer DP algorithms can overcome many of the limitations of their iterative counterparts. While iterative algorithms often have poor or no temporal locality, recursive algorithms have excellent and often optimal temporal locality. One problem with recursive

[1]unless specified otherwise, we will use "processor" and "processing core" synonymously

```
PAR-LOOP-PARENTHESIS(C, n)
(1) for t ← 2 to n − 1 do
(2)   parallel for i ← 1 to n − t do
(3)     j ← t + i
(4)     for k ← i + 1 to j do
(5)       C[i, j] ← min (C[i, j], C[i, k] + C[k, j] + w(i, k, j))
```

Figure 1: Parallel iterative algorithm for solving the parenthesis problem (a.k.a. the matrix-chain multiplication problem) which computes an optimal way to parenthesize a sequence of n matrices so that the cost of multiplying them is minimum.

divide-and-conquer algorithms is that they trade off parallelism for cache optimality, and thus may end up with suboptimal parallelism.

For example, a 2-way recursive algorithm (where, each dimension of the subtask will be half the dimension of its parent task) for the parenthesis problem has $T_\infty(n) = \Theta\left(n^{\log_2 3}\right)$ and $Q_1(n) = \Theta\left(n^3/(B\sqrt{M})\right)$, that is, it has optimal serial cache complexity but suboptimal span [49]. For n-way recursive algorithm, $T_\infty(n) = \Theta(n \log n)$ and $Q_1(n) = O\left(n^3\right)$. This time, the span is almost linear in n but the serial cache complexity is the worst possible. Ideally, we want to have a balance between cache complexity and span by choosing r-way recursive algorithm in which case neither the span nor the parallel cache complexity will be the best possible, however, will have best practical performance.

Source of suboptimal parallelism in recursive algorithms. The suboptimal parallelism in 2-way recursive algorithms results from artificial dependencies among subproblems that are not implied by the underlying DP recurrence [47]. We use the 2-way recursive divide-and-conquer algorithm for the *Longest Common Subsequence* (LCS) problem as an example below.

For the LCS problem, each cell (i, j) in the DP table depends on a cell to its left $(i, j-1)$, a cell above $(i-1,$ j) and a cell on the diagonal $(i-1, j-1)$. The 2-way recursive divide-and-conquer algorithm for the LCS problem splits the DP table X into four equal quadrants: X_{11} (top-left), X_{12} (top-right), X_{21} (bottom-left), and X_{22} (bottom-right). It then recursively computes the quadrants in a way that respects the cell dependencies among the quadrants: first X_{11}, then X_{12} and X_{21} in parallel, and finally X_{22}. Notice that, the top-left quadrants of X_{12} and X_{21} i.e., $X_{12,11}$ and $X_{21,11}$, respectively, can only start executing when the execution of the bottom-right quadrant of X_{11} i.e., $X_{11,22}$ completes. These dependencies amomg subtasks are not implied by the DP recurrence but arise from the recursive structure of the algorithm. $X_{12,11}$ (resp. $X_{21,11}$) can start executing as soon as $X_{11,12}$ (resp. $X_{11,21}$) is done. We call these dependencies *artificial dependencies* and they appear at several different granularities. Most often, these artificial dependencies asymptotically increase the span, thereby reducing parallelism.

Recursive wavefront algorithms. By removing artificial dependencies from the recursive algorithms, it is possible to develop algorithms that simultaneously achieve parallel cache-optimality, near-optimal parallelism, and cache-obliviousness. Such algorithms are called *recursive wavefront* (or *cache-oblivious wavefront*) *algorithms*.

The recursive wavefront algorithms were introduced in [47]. However, those algorithms (also called COW algorithms) are too complicated to develop, analyze, implement, and generalize. Atomic

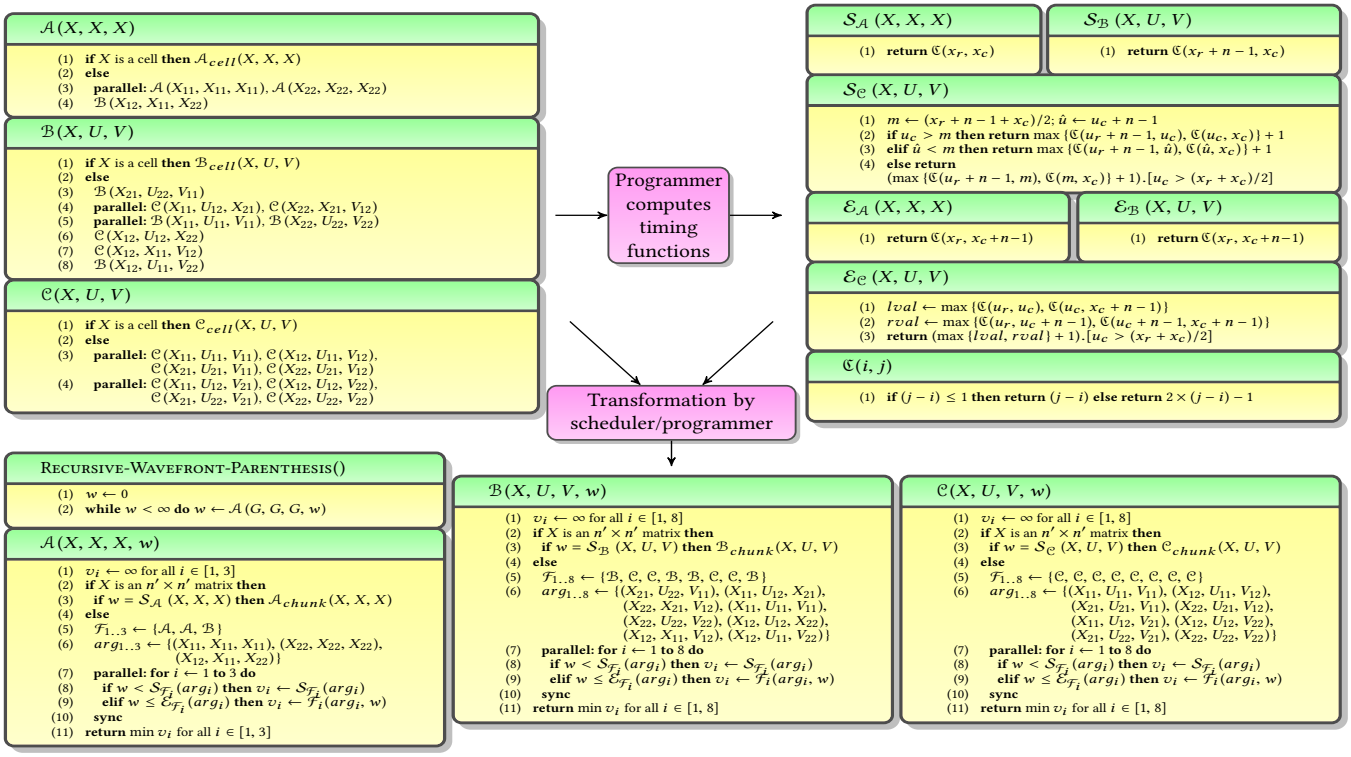

Figure 2: TOP: The programmer derives the timing functions (top-right) from a given standard 2-way recursive divide-and-conquer DP algorithm for the parenthesis problem (top-left). A region Z has its top-left corner at (z_r, z_c) and is of size $n \times n$. BOTTOM: A recursive divide-and-conquer wavefront algorithm is generated for the parenthesis problem. The programmer derives the algorithm if work-stealing scheduler (see Section 4.1) is used and the scheduler derives the algorithm if W-SB scheduler (see Section 4.2) is used. The algorithm makes use of the timing functions derived by the programmer.

operations were used to identify and launch ready tasks, and implementations required hacking into Cilk's runtime system. No bounds on parallel cache complexities of those algorithms are known.

In this paper, we present a generic method to schedule recursive wavefront algorithms based on timing functions. These algorithms have a structure similar to the standard recursive divide-and-conquer algorithms, but each recursive function call is annotated with start-time and end-time hints that are passed to the scheduler. The task scheduler will make sure that the algorithms are executed in a wavefront fashion using the timing functions. Indeed, the actions the scheduler is expected to take based on the timing functions is straightforward, and a programmer may choose to make some straightforward transformations of the code herself and use a scheduler that does not accept hints. The transformed code is still purely based on fork-join parallelism, and the performance bounds (e.g., parallel running time and parallel cache complexity) guaranteed by any scheduler supporting fork-join parallelism apply. The recursive wavefront algorithm for the parenthesis problem has $T_\infty(n) = \Theta(n \log n)$ and $Q_1(n) = O\left(n^3/(B\sqrt{M})\right)$. Bounds on T_p and Q_p can be obtained from scheduler guarantees.

Related work. The tiled iterative algorithms [20, 29, 37, 41, 43, 53] have been studied extensively as tiling is the traditional way of implementing dynamic programming and other matrix algorithms. There are several frameworks to automatically produce tiled codes such as PLuTo [10], Polly [30], and PoCC [38]. However, these software programs are not designed to generate efficient parallel tiled code for non-trivial DP recurrences. The major concerns with tiled programs are that they are cache-aware and sometimes processor-aware that sacrifices portability across machines. Another disadvantage of being cache-aware is that the algorithms are not cache-adaptive [7], i.e., the algorithms do not adapt to changes in available shared cache/memory space during execution and hence may run slower when multiple programs run concurrently in a shared-memory environment [14, 48]. Several existing systems such as Bellman's GAP compiler [28], semi-automatic synthesizer [39], EasyPDP [44], EasyHPS [21], pattern-based system [36], and parallelizing plugins [40] can be used to generate iterative and tiled loop programs. Parallel task graph execution systems such as Nabbit [5] and BDDT [50] execute DP tasks through unrolling, and lack cache efficiency.

The classic 2-way recursive divide-and-conquer algorithms having optimal serial cache complexity and good (but, not always optimal) parallelism have been developed, analyzed, and implemented in [16, 17, 49]. Hybrid r-way algorithms are considered in [16] but they are either cache- or processor-aware and complicated to program. Pochoir [45] is used to generate cache-oblivious implementations of stencil computations. However, the recursive algorithms often have low parallelism due to artificial dependencies among subtasks. Recently Aga et al. in [4] proposed a speculation approach to alleviate the concurrency constraints imposed by the

artificial dependencies in standard parallel recursive divide-and-conquer programs and reported up to a 1.6× speedup over their baseline on 30 cores.

The recursive wavefront algorithms were introduced in [47] but they are complicated to develop, analyze, implement, and generalize. They make extensive use of atomic instructions, and standard analysis model of fork-join parallelism does not apply. In this paper, we try to address these issues.

Our contributions. Our major contributions are as follows:
(1) [**Algorithmic**.] We present a generic method to develop and schedule recursive wavefront algorithms based on timing functions. We present two approaches for scheduling a recursive wavefront algorithm: (*i*) the algorithm passes timing functions and space usage info to a hint-accepting space-bounded scheduler, and (*ii*) the programmer appropriately transforms the algorithm to use the timing functions, and uses a standard randomized work-stealing scheduler to run the program.
(2) [**Experimental**.] We present performance and scalability results of the presented algorithms on state-of-the-art multicore machines and show a comparative analysis with standard 2-way recursive divide-and-conquer and the original cache-oblivious wavefront (COW) algorithms from [47].

2 DERIVING RECURSIVE WAVEFRONT ALGORITHMS

In this section, we describe how to transform a standard recursive DP algorithm into a recursive wavefront algorithm. We have shown very recently that for a wide class of DP problems which includes the LCS problem, the parenthesis problem and Floyd-Warshall's all pair shortest path (APSP) problems among others, standard 2-way recursive DP algorithms can be generated automatically from simple iterative descriptions of the underlying DP recurrences [14, 33]. Our transformation [27] involves augmenting all recursive function calls with timing functions to launch them as early as possible without violating any dependency constraints implied by the DP recurrence. The timing functions are derived analytically and do not employ locks or atomic instructions.

Our transformation allows the updates to the DP table proceed in an order close to iterative wavefront, but from within the structure of a recursive divide-and-conquer algorithm. The goal is to reach the higher parallelism of an iterative wavefront algorithm while retaining the better cache performance (i.e., efficiency and adaptivity) and portability (i.e., cache- and processor-obliviousness) of a recursive algorithm [48, 49].

Wavefront Order. Let us first define the *wavefront order* of applying updates to a DP table. Each update writes to one DP table cell by reading values from other cells. We say that a cell is *fully updated* provided it is never updated in the future. An update becomes *ready* when all cells it reads from are fully updated. We assume that only ready updates can be applied and each such update can only be applied once. A wavefront order of updates proceeds in discrete timesteps. In each step, all ready updates to distinct cells are applied in parallel. However, if a cell has multiple ready updates only one of them is applied, and the rest are retained for future. A wavefront order does not have any artificial dependencies.

Transformation. It is completed in three major steps:
(1) [**Construct completion-time function**.] A closed-form formula is derived based on the original DP recurrence that gives the timestep at which each DP cell is fully updated in the wavefront order. See Section 2.1.
(2) [**Construct start- and end-time functions**.] Cell completion times are used to derive closed-form formulas that give the timesteps at which each recursive function call should start and end execution in the wavefront order. See Section 2.2.
(3) [**Derive the recursive wavefront algorithm**.] Each recursive function call in the standard recursive algorithm is augmented with its start- and end-time functions so that the algorithm can be used to apply the updates in any given timestep in wavefront order. We then use a variant of iterative deepening on top of this recursive algorithm to execute all timesteps efficiently. See Section 2.3.

We describe our transformation for arbitrary d-dimensional ($d \geq 1$) DP in which each dimension of the DP table is of the same length and is a power of 2.

Running example. We explain our transformation approach by applying it on a recursive algorithm for the parenthesis (a.k.a. matrix-chain multiplication) problem [12], which is defined as follows. Let $G[i, j]$ denote the minimum cost of parenthesizing $s_i \cdots s_j$. Then the 2D DP table $G[0 : n, 0 : n]$ is filled up as follows.

$$G[i,j] = \begin{cases} \infty & \text{if } 0 \leq i = j \leq n, \\ v_j & \text{if } 0 \leq i = j-1 < n, \\ \min\limits_{i \leq k \leq j} \left\{ \begin{matrix} G[i,k] \\ +G[k,j] \\ +w(i,k,j) \end{matrix} \right\} & \text{if } 0 \leq i < j-1 < n; \end{cases} \quad (1)$$

where the v_j's and function $w(\cdot,\cdot,\cdot)$ are given. The recurrence is evaluated by the recursive algorithm [49] given at the top-left corner of Figure 2. Most simple DP examples (including LCS) do not consider multiple functions, or do not have race conditions, or do not portray all possible read-write constraints. We use the parenthesis problem because it is one of the simplest (and well-known) DP problems which covers most of the issues we would like to explain here. In the rest of the section, we show how a recursive wavefront algorithm (shown in Figure 2) can be derived from the given standard recursive algorithm.

Consider the standard 2-way recursive algorithm for the parenthesis problem given at the top-left corner of Figure 2. It has three functions that update the DP table. Initially, function $\mathcal{A}(G, G, G)$ is called, where G is the entire DP table. Then the computation progresses by recursively breaking the table into quadrants and calling functions $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ on these smaller regions of G. At the base case (i.e., a 1×1 region of G), each function updates a cell. When x is a cell, function $\mathcal{A}(x, x, x)$ updates x by reading x itself which corresponds to the case $i = k = j$ in the recurrence. Similarly, $\mathcal{B}(x, u, v)$ updates cell x by reading x itself and two other cells u and v which correspond to cases $i = k \neq j$ and $i \neq k = j$. Finally, $\mathcal{C}(x, u, v)$ updates the cell x by reading the two cells u and v which corresponds to $i \neq k \neq j$.

The top part of Figure 3 shows how the standard 2-way recursive algorithm with 1×1 base case updates $G[1 : n, 1 : n]$ when $n = 8$. We use $\mathcal{F}_t$ in a cell to denote that function $\mathcal{F}$ updates the cell at timestep t, where $\mathcal{F} \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}\}$. Using an unbounded number of processors the standard recursive algorithm updates the entire table in 31 timesteps. In contrast, the bottom part of Figure 3 shows that an iterative wavefront algorithm will update G in only 18 timesteps.

Problem	Work (T_1)	Classic 2-way Recursive: Serial cache complexity (Q_1)	Classic 2-way Recursive: Span (T_∞)	Recursive Wavefront (this paper): Best serial cache complexity (Q_1)	Recursive Wavefront (this paper): Best span (T_∞)
Parenthesis problem [16]	$\Theta\left(n^3\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n^{\log_2 3}\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n \log n\right)$
Floyd-Warshall's APSP 3-D [17]	$\Theta\left(n^3\right)$	$\Theta\left(n^3/B\right)$	$\Theta\left(n \log^2 n\right)$	$\Theta\left(n^3/B\right)$	$\Theta\left(n \log n\right)$
Floyd-Warshall's APSP 2-D [17]	$\Theta\left(n^3\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n \log^2 n\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n \log n\right)$
LCS / Edit distance [15]	$\Theta\left(n^2\right)$	$\Theta\left(n^2/(BM)\right)$	$\Theta\left(n^{\log_2 3}\right)$	$\Theta\left(n^2/(BM)\right)$	$\Theta\left(n \log n\right)$
Gap problem [13]	$\Theta\left(n^3\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n^{\log_2 3}\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n \log n\right)$
3-point stencil	$\Theta\left(n^2\right)$	$\Theta\left(n^2/(BM)\right)$	$\Theta\left(n^{\log_2 3}\right)$	$\Theta\left(n^3/(BM)\right)$	$\Theta\left(n \log n\right)$
Protein accordion folding [49]	$\Theta\left(n^3\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n \log n\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n \log n\right)$
Spoken-word recognition [42]	$\Theta\left(n^2\right)$	$\Theta\left(n^2/(BM)\right)$	$\Theta\left(n^{\log_2 3}\right)$	$\Theta\left(n^2/(BM)\right)$	$\Theta\left(n \log n\right)$
Function approximation	$\Theta\left(n^3\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n^{\log_2 3}\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n \log n\right)$
Binomial coefficient [35]	$\Theta\left(n^2\right)$	$\Theta\left(n^2/(BM)\right)$	$\Theta\left(n^{\log_2 3}\right)$	$\Theta\left(n^2/(BM)\right)$	$\Theta\left(n \log n\right)$
Bitonic traveling salesman [19]	$\Theta\left(n^2\right)$	$\Theta\left(n^2/(BM)\right)$	$\Theta\left(n \log n\right)$	$\Theta\left(n^2/(BM)\right)$	$\Theta\left(n \log n\right)$

Table 2: Work (T_1), serial cache complexity (Q_1), span (T_∞), and parallelism (T_1/T_∞) of classic 2-way recursive and recursive wavefront algorithms for several DP problems. Here, n = problem size, M = cache size, B = block size, and p = #cores. We assume that the DP table is too large to fit into the cache, and $M = \Omega\left(B^d\right)$ when $\Theta\left(n^d\right)$ is the size of the DP table. On p cores, the running time is $T_p = O\left(T_1/p + T_\infty\right)$. All algorithms have a parallel cache complexity of $Q_p = O\left(Q_1 + p(M/B)T_\infty\right)$ w.h.p. when run under the randomized work-stealing scheduler on a parallel machine with private caches. The recursive wavefront algorithms have a parallel cache complexity of $Q_p = O\left(Q_1\right)$ when run under the modified space-bounded scheduler of Section 4.2. For each recursive wavefront algorithm we have listed the best Q_1 and the best T_∞ it can achieve though it may not achieve both simultaneously (depends on base case size, i.e., value of n' chosen). In Section 4 we discuss how a cache-oblivious recursive wavefront algorithm can match the asymptotic Q_1 of the 2-way recursive non-wavefront algorithm while asymptotically improving over the T_∞ of that non-wavefront algorithm (and thus getting closer to the T_∞ of the iterative wavefront algorithm).

Our recursive wavefront algorithm (shown at the bottom of Figure 2) can be viewed as a hybrid between the pure iterative wavefront algorithm and standard 2-way recursive algorithm (assuming that the algorithm switches to standard 2-way recursive algorithm when it reaches a base case of size $n' \times n'$). Indeed, with a 1×1 base case our recursive wavefront algorithm will perform the updates in exactly the same order as the highly parallel iterative wavefront algorithm and terminate in 18 steps, and with an $n \times n$ base case it will reduce to the highly cache-efficient standard 2-way recursive algorithm. However, we are more interested in base case sizes that lie between these two extremes and results in useful tradeoffs between parallelism and cache performance.

2.1 Constructing completion-time function

This section defines completion-time, and shows how to compute it in $O\left(1\right)$ time for any cell.

*Definition 2.1 (**Completion-time**).* The completion-time $\mathfrak{C}(x)$ for cell x is the timestep in wavefront order at which x is fully updated. More formally, $\mathfrak{C}(x) = \max\ t \mid$ for all $\mathcal{F}_t(x, \ldots)$, where $\mathcal{F}_t(x, \ldots)$ means that cell x is updated by function $\mathcal{F}$ at timestep t.

Figure 3(a–b) show cell completion-times when the standard 2-way recursive algorithm (3(a)) and the iterative wavefront algorithm (3(b)) for solving the parenthesis problem are run on an 8×8 DP table. Observe that while all cells are updated in 31 steps in part 3(a), in part 3(b) they are completed in only 18 timesteps. Figure 3(c) shows cell update and completion times under the recursive wavefront algorithm using fractional timesteps. When $k > 0$, if multiple updates to a cell x are ready at integer time step t, this algorithm applies them one at a time at fractional timesteps $t.0, t.1, \ldots, t.(k-1)$ to avoid race conditions. Observe that the number of distinct fractional timesteps used is still exactly 18 as in the case of iterative wavefront algorithm (3(b)). Use of fractional timesteps makes completion times and update times easier to find which we explain below and in Section 2.2.

The completion-time function $\mathfrak{C}(x)$ will only return the integer part of the fractional timestep at which x is fully updated. The fractional part will be added back by the start-time and end-time functions of Section 2.2. We also assume that any cell x will be updated by a function of the form $\mathcal{F}(x, \ldots, y, \ldots)$ with $y = x$ at most once. We will call such an update a *self-update*.

Completion-time of a cell can be computed from the given DP recurrence as follows: $\mathfrak{C}(x) = smax(x) + su(x) + flag(x)$, where $smax(x)$ is the maximum completion time of the cells on which x directly depends, i.e., $smax(x) = \max_{\mathcal{F}(x,\ldots,y,\ldots) \wedge y \neq x} \mathfrak{C}(y)$; $su(x) = 1$ if x undergoes self-update, and 0 otherwise; and $flag(x) = 0$ if an update $\mathcal{F}(x, \ldots, y, \ldots)$ with $\mathfrak{C}(y) = smax(x)$ for some $y \neq x$ is a self-update for x, and 1 otherwise.

The completion-time for any cell (i, j) in the DP table for the parenthesis problem can be found as follows:

$$\mathfrak{C}(i,j) = \begin{cases} 0 & \text{if } i = j, \\ \mathfrak{C}(i, j-1) + 1 + 0 & \text{if } i = j-1, \\ \mathfrak{C}(i, j-1) + 1 + 1 & \text{if } i < j-1; \end{cases} \tag{2}$$

because $smax(i,j) = \mathfrak{C}(i,j-1) = \mathfrak{C}(i+1,j)$; for $i \leq j-1$, $su(i,j) = 1$ as cell (i,j) is updated by the self-update function $\mathcal{B}$, and $su(i,j) = 0$ otherwise; and for $i < j-1$, $flag(i,j) = 1$ as function $\mathcal{B}$ does not read from a cell $y \neq (i,j)$ with $\mathfrak{C}(y) = smax(i,j)$, and $flag(i,j) = 0$ otherwise. Solving the recurrence (assuming that race will be avoided by fractional timing as explained in Section 2.2), we get the following: $\mathfrak{C}(i,j) = 0$ if $i = j$ and $\mathfrak{C}(i,j) = 2(j-i) - 1$ if $i < j$.

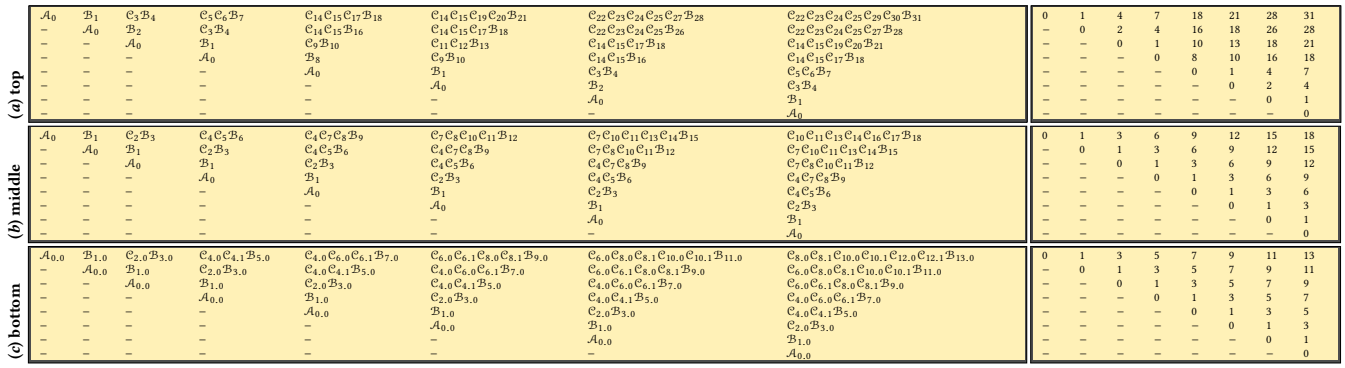

Figure 3: Timesteps at which each DP table cell is updated ($\mathcal{F}_t$ means function $\mathcal{F}$ updates at timestep t) and the timestep at which each cell becomes fully updated (on the right) for the parenthesis problem on a DP table of size 8×8 using (a) top: standard 2-way recursive algorithm, (b) middle: iterative wavefront algorithm, and (c) bottom: recursive wavefront algorithm with fractional timesteps. Observe that the number of fractional timesteps in the recursive wavefront algorithm (bottom part) is exactly the same as that in the iterative wavefront algorithm (middle part). Both recursive algorithms use a 1×1 base case. We assume that the number of processors is unbounded.

2.2 Constructing start-time and end-time functions

In this section, we define start-time and end-time for a recursive function call, and show how to derive them from completion-times.

*Definition 2.2 (**Start-time, end-time**).* The start-time (resp. end-time) of a recursive function call in a recursive wavefront algorithm is the earliest (resp. latest) timestep in the wavefront order at which one of the updates to be applied by that function call (either directly or through a recursive function call) becomes ready.

Let $\mathcal{F}(X, Y_1, \ldots, Y_s)$ be a function call that writes to a region X by reading from regions $Y_1, \ldots, Y_s$ of the DP table. Its start- and end-times, denoted by $\mathcal{S}_{\mathcal{F}}(X, Y_1, \ldots, Y_s)$ and $\mathcal{E}_{\mathcal{F}}(X, Y_1, \ldots, Y_s)$, respectively, are computed as follows.

$$\underbrace{\mathcal{S}_{\mathcal{F}}(X, Y_1, \ldots, Y_s)}_{X \in \{Y_1, \ldots, Y_s\}} = \begin{cases} (\mathfrak{C}(X)).0 & \text{if } X \text{ is a cell,} \\ \min\ \mathcal{S}_{\mathcal{F}'}\left(X', Y_1', \ldots, Y_s'\right) & \text{otherwise;} \end{cases}$$

$$\underbrace{\mathcal{S}_{\mathcal{F}}(X, Y_1, \ldots, Y_s)}_{X \notin \{Y_1, \ldots, Y_s\}} = \begin{cases} (\max_{1 \le i \le s}\{\mathfrak{C}(Y_i)\} + 1) & \\ .ra(X, Y_1, \ldots, Y_s) & \text{if } X \text{ is a cell,} \\ \min\ \mathcal{S}_{\mathcal{F}'}\left(X', Y_1', \ldots, Y_s'\right) & \text{otherwise;} \end{cases}$$

$$\underbrace{\mathcal{E}_{\mathcal{F}}(X, Y_1, \ldots, Y_s)}_{X \in \{Y_1, \ldots, Y_s\}} = \begin{cases} (\mathfrak{C}(X)).0 & \text{if } X \text{ is a cell,} \\ \max\ \mathcal{E}_{\mathcal{F}'}\left(X', Y_1', \ldots, Y_s'\right) & \text{otherwise;} \end{cases}$$

$$\underbrace{\mathcal{E}_{\mathcal{F}}(X, Y_1, \ldots, Y_s)}_{X \notin \{Y_1, \ldots, Y_s\}} = \begin{cases} (\max_{1 \le i \le s}\{\mathfrak{C}(Y_i)\} + 1) & \\ .ra(X, Y_1, \ldots, Y_s) & \text{if } X \text{ is a cell,} \\ \max\ \mathcal{E}_{\mathcal{F}'}\left(X', Y_1', \ldots, Y_s'\right) & \text{otherwise;} \end{cases}$$

where, in the non-cellular case, minimization/maximization is taken over all functions $\mathcal{F}'(X', Y_1', \ldots, Y_s')$ called by $\mathcal{F}(X, Y_1, \ldots, Y_s)$ recursively. Also, $ra(X, Y_1, \ldots, Y_s)$ is the problem-specific race avoidance condition used when two functions write to the same region. Though we use fractional timesteps for simplicity, the total number of distinct timesteps remain exactly the same as that in the iterative wavefront algorithm.

For the parenthesis problem, the start-times for the three functions $\mathcal{A}, \mathcal{B}$, and $\mathcal{C}$ are computed as below. Let (x_r, x_c), (u_r, u_c), and (v_r, v_c) denote the positions of the top-left cells of regions X, U and V, respectively. Then

$$\mathcal{S}_{\mathcal{A}}(X, X, X) = \begin{cases} \mathfrak{C}(X).0 & \text{if } X \text{ is a cell,} \\ \mathcal{S}_{\mathcal{A}}(X_{11}, X_{11}, X_{11}) & \text{otherwise;} \end{cases}$$

$$\mathcal{S}_{\mathcal{B}}(X, U, V) = \begin{cases} \mathfrak{C}(X).0 & \text{if } X \text{ is a cell,} \\ \mathcal{S}_{\mathcal{B}}(X_{21}, U_{22}, V_{11}) & \text{otherwise;} \end{cases}$$

$$\mathcal{S}_{\mathcal{C}}(X, U, V) = \begin{cases} (\max\{\mathfrak{C}(U), \mathfrak{C}(V)\} + 1) & \\ \cdot \left[u_c > \frac{x_r + x_c}{2}\right] & \text{if } X \text{ is a cell,} \\ \min\left\{\begin{matrix} \mathcal{S}_{\mathcal{C}}(X_{21}, U_{21}, V_{11}), \\ \mathcal{S}_{\mathcal{C}}(X_{21}, U_{22}, V_{21}) \end{matrix}\right\} & \text{otherwise;} \end{cases}$$

where [ ] is the *Iverson bracket* [34].

Both $\mathcal{A}$ and $\mathcal{B}$ read from and write to X, and hence their start-times follow directly from the first recurrence in Definition 2.2. In case of $\mathcal{B}$, X is updated by reading from pair $\langle U, X\rangle$ and also from $\langle X, V\rangle$. Function $\mathcal{C}$ follows the second recurrence from the definition. As $\mathcal{C}$ writes to the same region twice, there is a race and to avoid it we use the condition $[u_c > (x_r + x_c)/2]$ derived manually.

The end-times can be computed similarly. Solving the recurrences for the start-times and end-times, we obtain the timing functions shown in Figure 2.

2.3 Deriving a recursive wavefront algorithm

In this section, we describe how to use timing functions to derive a recursive wavefront algorithm from a given standard recursive divide-and-conquer DP algorithm. We use the parenthesis problem as an example.

A standard recursive algorithm for the parenthesis problem is shown at the top-left corner of Figure 2. We modify it as follows, and the modified algorithm is shown on the bottom part of the same figure.

First, we modify each function $\mathcal{F}$ to include a switching point $n' \ge 1$, and switch to the original non-wavefront recursive algorithm by calling $\mathcal{F}_{chunk}$ when the size of each input submatrix drops to $n' \times n'$ or below.

We augment each function to accept a timestep parameter w. We remove all serialization among recursive function calls by making sure that all functions that are called are launched in parallel. We do not launch a function unless w lies between its start-time and

end-time which means that a function is not invoked if we know that it does not have an update to apply at timestep w in wavefront order. Observe that the function $\mathcal{F}_{chunk}$ at switching does not accept a timestep parameter, but if we reach it we know that it has an update to apply at timestep w. However, once we enter that function we do not stop until we apply all updates that function can apply at all timesteps $\geq w$.

Each function is also modified to return the smallest timestep above w for which it may have at least one update that is yet to be applied. It finds that timestep by checking the start-time of each function that was not launched because the start-time was larger than w, and the timestep returned by each recursive function that was launched and taking the smallest of all of them.

Finally, we add a loop (see Recursive-Wavefront-Parenthesis in Figure 2) to execute all timesteps of the wavefront using the modified functions. We start with timestep $w = 0$, and invoke the main function $\mathcal{A}(G, G, G, w)$ which applies all updates at timestep w and depending on the value chosen for n' possibly some updates above timestep w, and returns the smallest timestep above w for which there may still be some updates that are yet to be applied. We next call function $\mathcal{A}$ with that new timestep value, and keep iterating in the same fashion until we are able to exhaust all timesteps.

3 APPLICATIONS

In this section, we present the recursive wavefront algorithms for LCS, Floyd-Warshall's APSP, and gap problem [48]. We will only give the timing functions and not the entire recursive wavefront algorithm. We give references to the papers that present the standard (non-wavefront) recursive algorithms from which recursive wavefront algorithms can easily be derived by plugging in the timing functions as per Section 2.3.

Longest common subsequence (LCS). The LCS problem [15, 32] asks one to find the longest of all common subsequences [19] between two strings. In LCS DP, a cell depends on its three adjacent cells. Here, we are interested in finding only the length of the LCS.

We build on the recursive algorithm given in [15] which has only one function $\mathcal{A}$ (i.e., named LCS-Output-Boundary in [15]). The timing functions are as follows.

$$\mathfrak{C}(i,j) = \begin{cases} 0 & \text{if } i < 0 \mid\mid j < 0 \mid\mid i = j = 0, \\ \max\begin{pmatrix} \mathfrak{C}(i-1,j), \mathfrak{C}(i,j-1), \\ \mathfrak{C}(i-1,j-1) \end{pmatrix} + 1 & \text{otherwise.} \end{cases}$$

$$\begin{aligned} \mathcal{S}_{\mathcal{A}}(X) = \mathcal{E}_{\mathcal{A}}(X) &= (\mathfrak{C}(X)).0 && \text{if } X \text{ is a cell,} \\ \mathcal{S}_{\mathcal{A}}(X) &= \mathcal{S}_{\mathcal{A}}(X_{11}) && \text{if } X \text{ is not a cell,} \\ \mathcal{E}_{\mathcal{A}}(X) &= \mathcal{E}_{\mathcal{A}}(X_{22}) && \text{if } X \text{ is not a cell.} \end{aligned}$$

Solving the recurrences, we have $\mathfrak{C}(i,j) = i+j$, $\mathcal{S}_{\mathcal{A}}(X) = \mathfrak{C}(x_r, x_c)$, and $\mathcal{E}_{\mathcal{A}}(X) = \mathfrak{C}(x_r+n-1, x_c+n-1)$, where, (x_r, x_c) is the top-left corner of X.

Gap problem. Sequence alignment with general gap penalty [25, 26, 49, 52] is a generalization of the edit distance problem. We build on the recursive algorithm given in [49]. The timing functions are as follows.

$$\mathfrak{C}(i,j) = \begin{cases} 0 & \text{if } i = -1 \mid\mid j = -1 \mid\mid i = j = 0, \\ \max(\mathfrak{C}(i-1,j), \mathfrak{C}(i,j-1)) + 2 & \text{otherwise.} \end{cases}$$

$$\mathcal{S}_{\mathcal{A}}(X,X) = \begin{cases} (\mathfrak{C}(X)).0 & \text{if } X \text{ is a cell,} \\ \mathcal{S}_{\mathcal{A}}(X_{11}, X_{11}) & \text{otherwise.} \end{cases}$$

$$\mathcal{S}_{\mathcal{B}}(X,U) = \begin{cases} (\mathfrak{C}(U)+1).0 & \text{if } X \text{ is a cell,} \\ \mathcal{S}_{\mathcal{B}}(X_{11}, U_{11}) & \text{otherwise.} \end{cases}$$

$$\mathcal{S}_{\mathcal{C}}(X,V) = \begin{cases} (\mathfrak{C}(V)+1).[x_c \geq 1] & \text{if } X \text{ is a cell,} \\ \mathcal{S}_{\mathcal{C}}(X_{11}, V_{11}) & \text{otherwise.} \end{cases}$$

Function $\mathcal{B}$ does not have races and hence its $ra(X,U) = 0$. But when we add function $\mathcal{C}$, there is a race condition with $\mathcal{B}$ when the completion-time of read cell of $\mathcal{C}$ is the same as the completion-time of one of the cells on the left of the write cell of $\mathcal{C}$. To avoid clashes with $\mathcal{B}$, we use $ra(X,V) = [x_c \geq 1]$. We can write similar recurrence for the end-times. Solving the recurrences, we have

$$\begin{aligned} &\mathfrak{C}(i,j) = 2(i+j), \\ &\mathcal{S}_{\mathcal{F}}(X,Y) = (\mathfrak{C}(y_r, y_c) + [X \neq Y]).ra(X,Y), \\ &\mathcal{E}_{\mathcal{F}}(X,Y) = (\mathfrak{C}(y_r+n-1, y_c+n-1) + [X \neq Y]).ra(X,Y); \end{aligned}$$

where, when $\mathcal{F}$ is $\mathcal{A}$ (resp. $\mathcal{B}, \mathcal{C}$), then Y is X (resp. U, V); (y_r, y_c) is the top-left corner of region Y; and $ra(X,Y) = [x_c \geq 1]$ for function $\mathcal{C}$, and $ra(X,Y) = 0$, otherwise.

Floyd-Warshall's all-pairs shortest path (APSP). We build on the recursive algorithm for Floyd-Warshall's APSP [23, 51] given in [17]. However, that algorithm violates our assumption that cells can only be updated using values from fully updated cells. That violation can be removed by performing the computation in cubic space instead of quadratic space as explained in [14, 19]. We find the timing functions the cubic space version which remain valid for the DP using quadratic space.

The recursive wavefront algorithm can be easily derived using the start- and end-time functions for the recursive algorithm given in [17]. The recursive algorithm must be modified slightly to account for the third dimension through the variable k. Then, the completion-time function for an (i,j,k)-cell can be found from the DP recurrence as:

$$\mathfrak{C}(i,j,k) = \begin{cases} -1 & \text{if } k = -1, \\ \max\begin{pmatrix} \mathfrak{C}(i,j,k-1), \mathfrak{C}(i,k,k-1), \\ \mathfrak{C}(k,j,k-1)) + 1 \end{pmatrix} & \text{otherwise.} \end{cases}$$

From Definition 2.1, we have $smax(i,j,k) = \max(\mathfrak{C}(i,j,k-1), \mathfrak{C}(i,k,k-1), \mathfrak{C}(k,j,k-1))$ and $su(i,j,k) = 0$. Similarly, the start- and end-time can be found as follows. When X is a cell, the start- and end-times of all functions is $\mathfrak{C}(x_r, x_c, x_h)$. The recurrences for start-times are as follows.

$$\mathcal{S}_{\mathcal{A}}(X) = \begin{cases} \mathfrak{C}(x_r, x_c, x_h) & \text{if } X \text{ is a cell,} \\ \mathcal{S}_{\mathcal{A}}(X_{111}) & \text{otherwise.} \end{cases}$$

$$\mathcal{S}_{\mathcal{B}}(X,U) = \begin{cases} \mathfrak{C}(x_r, x_c, x_h) & \text{if } X \text{ is a cell,} \\ \mathcal{S}_{\mathcal{B}}(X_{111}, U_{111}) & \text{otherwise.} \end{cases}$$

$$\mathcal{S}_{\mathcal{C}}(X,V) = \begin{cases} \mathfrak{C}(x_r, x_c, x_h) & \text{if } X \text{ is a cell,} \\ \mathcal{S}_{\mathcal{C}}(X_{111}, V_{111}) & \text{otherwise.} \end{cases}$$

$$\mathcal{S}_{\mathcal{D}}(X,U,V) = \begin{cases} \mathfrak{C}(x_r, x_c, x_h) & \text{if } X \text{ is a cell,} \\ \mathcal{S}_{\mathcal{D}}(X_{111}, U_{111}, V_{111}) & \text{otherwise.} \end{cases}$$

Similarly, recurrences can be written for the end-times as well. Solving the recurrences, we have

$$\mathfrak{C}(i,j,k) = 3k + [i \neq k] + [j \neq k],$$

where, [ ] is the Iversion bracket. Let (x_r, x_c, x_h) be the cell with the smallest coordinates in X. Then for each $\mathcal{F} \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}\}$,

$$\mathcal{S}_{\mathcal{F}}(X,\ldots) = \mathfrak{C}(x_r, x_c, x_h)$$

$$\text{and } \mathcal{E}_{\mathcal{F}}(X,\ldots) = \max\left\{\begin{array}{c}\mathfrak{C}(x_r, x_c, x_h + n - 1),\\ \mathfrak{C}(x_r, x_c + n - 1, x_h + n - 1),\\ \mathfrak{C}(x_r + n - 1, x_c, x_h + n - 1),\\ \mathfrak{C}(x_r + n - 1, x_c + n - 1, x_h + n - 1)\end{array}\right\}$$

4 SCHEDULING RECURSIVE WAVEFRONT ALGORITHMS

In this section, we show how to schedule recursive wavefront algorithms to achieve provably good bounds (optimal or near-optimal) for both parallelism and cache performance.

Recall that our recursive wavefront algorithm switches to the original non-wavefront recursive algorithm when the input parameter n drops to a value $\leq n'$. While both recursive (wavefront and non-wavefront) algorithms have the same serial work complexity T_1, their spans are different. We use $T_\infty^R(n')$ to denote the span of the non-wavefront algorithm for a problem of size n'.

4.1 Using a work-stealing (WS) scheduler

In this section, we analyze the complexity of the hybrid recursive wavefront algorithms when scheduled using a randomized work-stealing (WS) scheduler [9].

Theorem 4.1 (**WS complexity**). *Suppose a DP recurrence is evaluated by the iterative wavefront algorithm in $N_\infty(n)$ parallel steps with an unbounded number of processors, where n is the size of each dimension of the DP table. When the recursive wavefront algorithm with switching point $n' \leq n$ evaluates the recurrence, we achieve the following bounds under a randomized work-stealing (WS) scheduler:*

(i) span, $T_\infty(n) = O\left(N_\infty(n/n') \times \left(\log n + T_\infty^R(n')\right)\right)$,

(ii) parallel time, $T_p(n) = O\left(T_1(n)/p + T_\infty(n)\right)$ (w.h.p. in n),

(iii) parallel cache complexity, $Q_p(n) = O\left(Q_1(n) + p\,(M/B)\,T_\infty(n)\right)$ (w.h.p. in n), and

(iv) extra space, $S_p(n) = O(p \log n)$.

We assume that $N_\infty(n)$ and $T_1(n)$ (work) are polynomials of n, and choose n' such that $T_1(n') = \Omega(\log n)$.

Proof. Observe that the outer loop (e.g., the loop inside Recursive-Wavefront-Parenthesis of Figure 2) in the recursive wavefront algorithm will iterate $N_\infty(n/n')$ times.

With an unbounded number of processors reaching the switching point requires $O(\log(n/n'))$ time and the span below that point is $T_\infty^R(n')$. The bound for $T_\infty(n)$ follows.

We reach the switching point $T_1(n/n')$ times. So the total work is $\Theta\left(T_1(n/n')\left(T_1(n') + \log(n/n')\right)\right)$. Since we assume $T_1(n)$ to be a polynomial function of n, we have $T_1(n/n') = \Theta\left(T_1(n)/T_1(n')\right)$. Then the total work remains $\Theta(T_1(n))$ assuming $T_1(n') = \Omega(\log n)$.

Let $T_1^{(i)}$, $T_\infty^{(i)}$ and $Q_1^{(i)}$ be the work, span and serial cache complexity, respectively, of the i-th iteration of the outer loop in the recursive wavefront algorithm. Then the parallel running time of that iteration under the WS scheduler is $O\left(1 + T_1^{(i)}(n)/p + T_\infty^{(i)}(n)\right)$ (w.h.p.). We sum up over all i, and obtain the claimed bound for $T_p(n)$.

The parallel cache complexity of the i-th iteration of the outer loop is $O\left(Q_1^{(i)}(n) + p(M/B)T_\infty^{(i)}(n)\right)$ (w.h.p.) under the WS scheduler. Summing up over all i gives us the claimed bound for $Q_p(n)$.

The bound on extra space comes from the observation that since we reduce n by a factor of 2 in each level of recursion the number of levels of recursion is $O(\log n)$. So each processor will require $O(\log n)$ stack space, and since there are p processors the bound on $S_p(n)$ follows. □

The following lemma gives a condition under which the recursive wavefront algorithm will have the same asymptotic serial cache complexity as the recursive non-wavefront algorithm.

Lemma 4.2 (**WS complexity**). *Suppose the original recursive non-wavefront algorithm has a serial cache complexity of $Q_1^R(n) = O\left(T_1(n)/f(M,B)\right)$, where n is size of each dimension of the DP table, f is a function of cache size M and line size B, $T_1(n)$ is a polynomial of n, and the size of the DP table is $\Omega(M)$. Then if the recursive wavefront algorithm with switching point $n' = n^\alpha$ for some $\alpha \in (0,1]$ evaluates the recurrence, it will achieve a serial cache complexity of $Q_1(n) = \Theta\left(Q_1^R(n)\right)$ provided the size of the DP (sub-)table corresponding to n' is $\Omega(M)$.*

The proof of the lemma above follows directly from the observation that the recursive wavefront algorithm will reach the switching point $\Theta\left(T_1(n)/T_1(n')\right)$ times, and each time will incur $Q_1^R(n') = O\left(T_1(n')/f(M,B)\right)$ serial cache misses. Thus its total serial cache complexity will be $\Theta\left(T_1(n)/T_1(n')\right) \times Q_1^R(n') = O\left(T_1(n)/f(M,B)\right)$.

Hence, if the original recursive non-wavefront algorithm has optimal serial cache complexity, the recursive wavefront algorithm will also have the same under the conditions given above.

4.2 Using a modified space-bounded (W-SB) scheduler

In this section, we show how to modify a space-bounded scheduler [18] so that it can execute a recursive wavefront algorithm cache-optimally with parallelism higher than that of the corresponding recursive non-wavefront algorithm.

For each recursive function call, our W-SB scheduler accepts three hints: start-time, end-time and working set size (i.e., the total size of all regions in the DP table accessed by the function call). Given an implementation of a standard recursive algorithm with each function call annotated with those three hints, the W-SB can automatically generate a recursive wavefront implementation (similar to the one at the bottom of Figure 2). From the given start-times, the scheduler determines the lowest start-time and executes the tasks that can be executed at that lowest start-time. Since the scheduler knows all the cache sizes, as soon as the working set size of any function executing on a processor under a cache fits into that cache, the scheduler anchors the function to that cache in the sense that all recursive function calls made by that function and its descendants will only be executed by the processors under that anchored cache. This approach of limiting migration of tasks ensures cache-optimality [8, 18].

Theorem 4.3 (**W-SB complexity**). *Suppose a DP recurrence is evaluated by the iterative wavefront algorithm in $N_\infty(n)$ parallel steps with an unbounded number of processors, where n is the size of each dimension of the DP table. When the recursive wavefront algorithm with switching point $n' \leq n$ evaluates the recurrence, we achieve the following bounds under the modified space-bounded (W-SB) scheduler:*

(i) span, $T_\infty(n) = O\left(N_\infty(n/n') \times \left(\log n + T_\infty^R(n')\right)\right)$,
(ii) parallel cache complexity, $Q_p(n) = O\left(Q_1(n)\right)$,
(iii) extra space, $S_p(n) = O\left(p \log n\right)$.

We assume that $N_\infty(n)$ and $N_1(n)$ (work) are polynomials of n, and choose n' such that $T_1^R(n') = \Omega(\log n)$.

Proof. The arguments for $T_\infty(n)$ and $S_p(n)$ are the same as those given in the proof of Theorem 4.1. The parallel cache complexity is found as follows.

When the working set size of a function call fits into a cache $\mathcal{M}$ the W-SB scheduler does not allow any recursive function calls made by that function or its descendants to migrate to other caches that are not a part of the subtree of caches rooted at $\mathcal{M}$. This implies the data is read completely and as much work as possible is done on this loaded cache data blocks in $\mathcal{M}$ before kicking them out of the cache. Hence, temporal cache locality is fully exploited at $\mathcal{M}$. As shown in [8, 18] being able to achieve cache-optimality for working set sizes that are smaller than the cache size by at most a constant factor guarantees $Q_p(n) = \Theta\left(Q_1(n)\right)$ for our algorithms. □

Cache-optimality is achieved under conditions similar to those given in Lemma 4.2.

5 EXPERIMENTAL RESULTS

In this section, we present experimental results showing the performance of recursive wavefront algorithms for the LCS, Parenthesis and the 2D FW-APSP problems. We also compare the performance of those algorithms with the corresponding standard 2-way recursive divide-and-conquer and the original cache-oblivious wavefront (COW) algorithms [47].

We used C++ with Intel© Cilk™ Plus extension to implement all algorithms presented in this section. Therefore, all implementations used the work-stealing scheduler provided by Cilk™ runtime system. All programs were compiled with `-O3 -ip -parallel -AVX -xhost` optimization parameters. In order to reduce the overhead of recursion and to take advantage of vectorization all implementations switch to an iterative kernel when n is sufficiently small (e.g., 64 for Parenthesis and Floyd-Warshall's APSP, and 256 for LCS). To measure cache performance we used PAPI-5.3[2]. Table 3 lists the systems on which we ran our experiments.

Model	E5-2680	E5-4650	E5-2680
Cluster	Stampede [3]	Stampede [3]	Comet [1]
#Cores	2x8	4x8	2x12
Frequency	2.70GHz	2.70GHz	2.50GHz
L1	32K	32K	32K
L2	256K	256K	256K
L3	20480K	20480K	30720K
Cache-line size	64B	64B	64B
Memory	64GB	1TB	64GB
Compiler	15.0.2	15.0.2	15.2.164
OS	CentOS 6.6	CentOS 6.6	CentOS 6.6

Table 3: System specifications.

Projected parallelism. we have used the Intel© Cilkview scalability analyzer to compute the ideal parallelism and burdened span of the following implementations: (*i*) recursive wavefront algorithm that does not switch to the 2-way non-wavefront recursive algorithm and instead directly uses an iterative basecase (`wave`), (*ii*) recursive wavefront algorithm that switches to the 2-way recursive divide-and-conquer at some point (`wave-hybrid`), (*iii*) standard 2-way recursive divide-and-conquer algorithm (`CO_2Way`). For `wave-hybrid`, we have used $n' = \max\{256, \text{power of 2 closest to } n^{2/3}\}$.

Figure 5 shows the ideal parallelism reported by Cilkview for `wave-hybrid`, `wave` and `CO_2Way` algorithms for solving LCS, Parenthesis and Floyd-Warshall's APSP problems. These plots show that recursive wavefront algorithms scale much better than standard 2-way recursive divide-and-conquer algorithms. For example, when solving the parenthesis problem for a 16k×16k matrix, though `CO_2Way` scales up to 23 processors only, `wave` and `wave-hybrid` scale up to 1916 and 823 processors, respectively.

Running time and cache performance. Figure 4 shows performance of the following on a 16-core Sandy Bridge machine: (*i*) `wave`, (*ii*) `wave-hybrid`, (*iii*) `CO_2Way`, and (*iv*) the original cache-oblivious wavefront (COW) algorithms with atomic operations [47]. Since we have already shown in [14, 49] that `CO_2Way` outperforms parallel iterative and blocked iterative codes for the parenthesis and Floyd-Warshall's APSP problems, we have not repeated those results here.

For `wave-hybrid`, we have used $n' = \max\{256, \text{power of 2 closest to } n^{2/3}\}$. Figure 4 clearly shows that `wave` and `wave-hybrid` algorithms perform better than `CO_2Way` and the COW algorithms for all cases. For parenthesis problem, `wave` is 2.6×, and `wave-hybrid` is 2× faster than `CO_2Way`. Similarly, the number of cache misses of `CO_2Way` is slightly higher than that of both `wave` and `wave-hybrid`. For LCS `wave` is 1.5×, and `wave-hybrid` is 1.7× faster than `CO_2Way` and cache miss trends follow along. For Floyd-Warshall's APSP, `wave` is 18%, and `wave-hybrid` is 10% faster than `CO_2Way`. Therefore, even with 16 cores, the impact of improvements in parallelism and cache-misses is visible on the running time. On the other hand, though COW algorithms have excellent theoretical parallelism, their implementations heavily use atomic operations, which may have impacted their performance negatively for large n, especially for DP dimension $d > 1$.

Figures 6 and 7 show performance results on a 24-core Haswell machine. Values for n' and size of iterative kernels were determined in the same way as before. For FW-APSP, `wave` is 15% and `wave-hybrid` is 10% faster than `CO_2Way`. Although we see improvements in L1 and L2 cache misses, the number of L3 misses were worse here, probably due to increased parallelism. For LCS problem, `wave` is 57%, `wave-hybrid` is 60% and the original COW algorithm is 26% faster than the `CO_2Way` implementation. For parenthesis problem, `wave` is 16% and `wave-hybrid` is 18% faster than `CO_2Way`, and we see only improvement in L3 misses.

On a Stampede node with 32-core Sandy Bridge processors, `wave` for FW-APSP runs 73% faster and `wave-hybrid` runs 69% faster than `CO_2Way`. On the other hand, for the parenthesis problem, both `wave` and `wave-hybrid` are 2.1× faster than `CO_2Way`.

To summarize, recursive wavefront algorithms are faster and more scalable than standard recursive divde and conquer algorithms for DP problems.

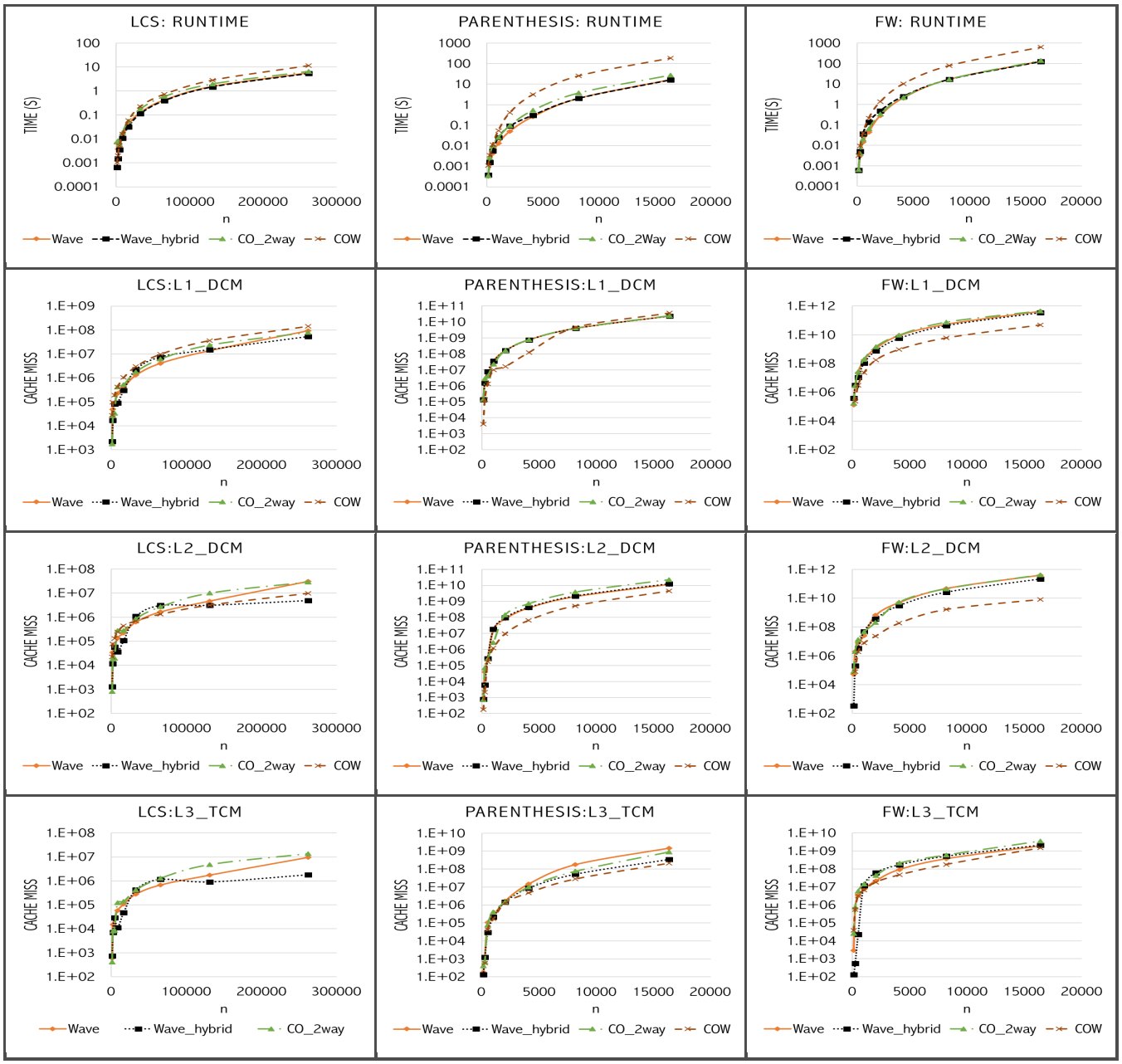

Figure 4: Runtimes and cache misses in three levels of caches for classic 2-way recursive divide-and-conquer, COW and recursive wavefront algorithms for LCS, Parenthesis and 2D FW-APSP Problems. All programs were run on 16 core machines in Stampede. All implementations used Cilk Plus's work-stealing scheduler.

6 CONCLUSION

We have presented a framework for designing recursive wavefront algorithms for dynamic programs which have excellent cache-complexity (i.e., temporal locality) and asymptotically more parallelism than standard 2-way recursive divide-and-conquer algorithms. The framework leads to theoretically fastest cache-oblivious parallel DP algorithms. Some open problems are as follows: (*i*) fully automate the framework, i.e., computation of timing functions and the race avoidance condition; (*ii*) investigate if recursive wavefront algorithms can achieve span asymptotically lower than $\Theta(n \log n)$; and (*iii*) extend the approach beyond DP problems.

ACKNOWLEDGMENTS

Chowdhury and Ganapathi were supported in part by NSF grants CCF-1439084 and CNS-1553510. Part of this work used the Extreme Science and Engineering Discovery Environment (XSEDE) which is supported by NSF grant ACI-1053575. The authors would like to thank anonymous reviewers for valuable comments and suggestions that have significantly improved the paper.

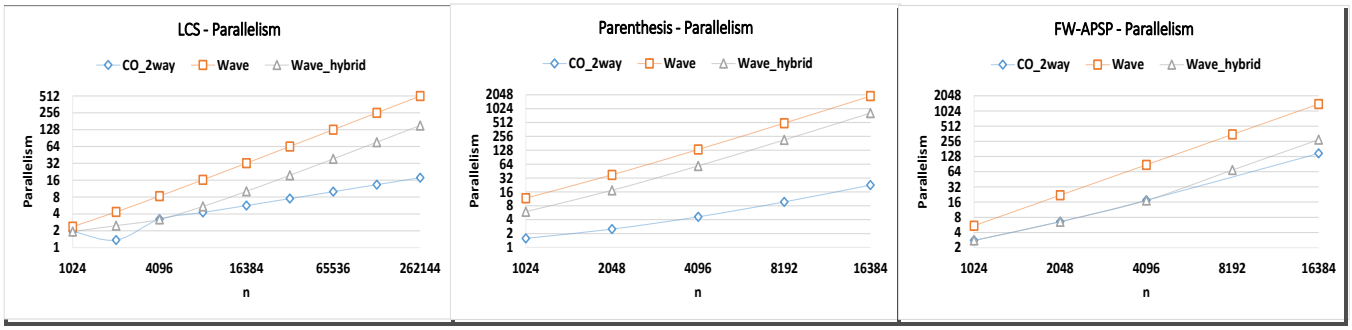

Figure 5: Projected scalability of new recursive wavefront algorithms by Cilkview Scalability Analyzer. The numbers denote till how many cores the implementation should scale linearly.

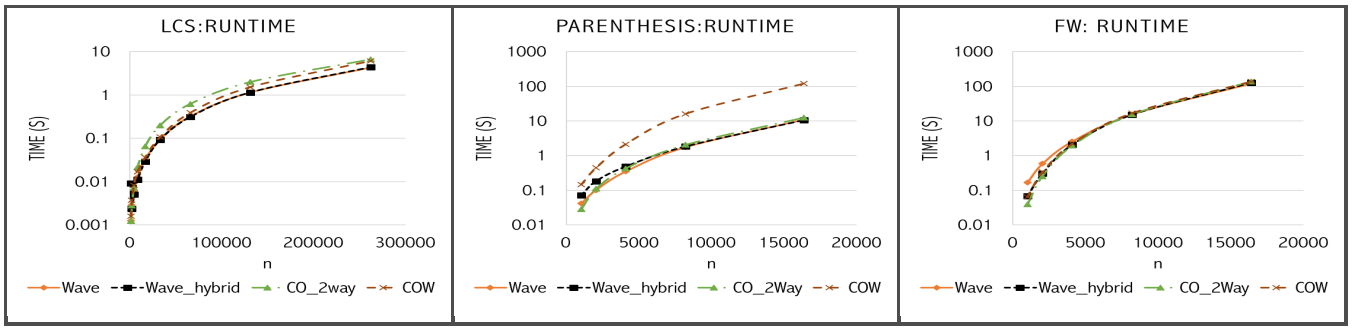

Figure 6: Runtimes for classic 2-way recursive divide-and-conquer, COW and recursive wavefront algorithms for LCS, Parenthesis and 2D FW-APSP. All programs were run on 24 core machines in Comet. All implementations used cilk plus's work-stealing scheduler.

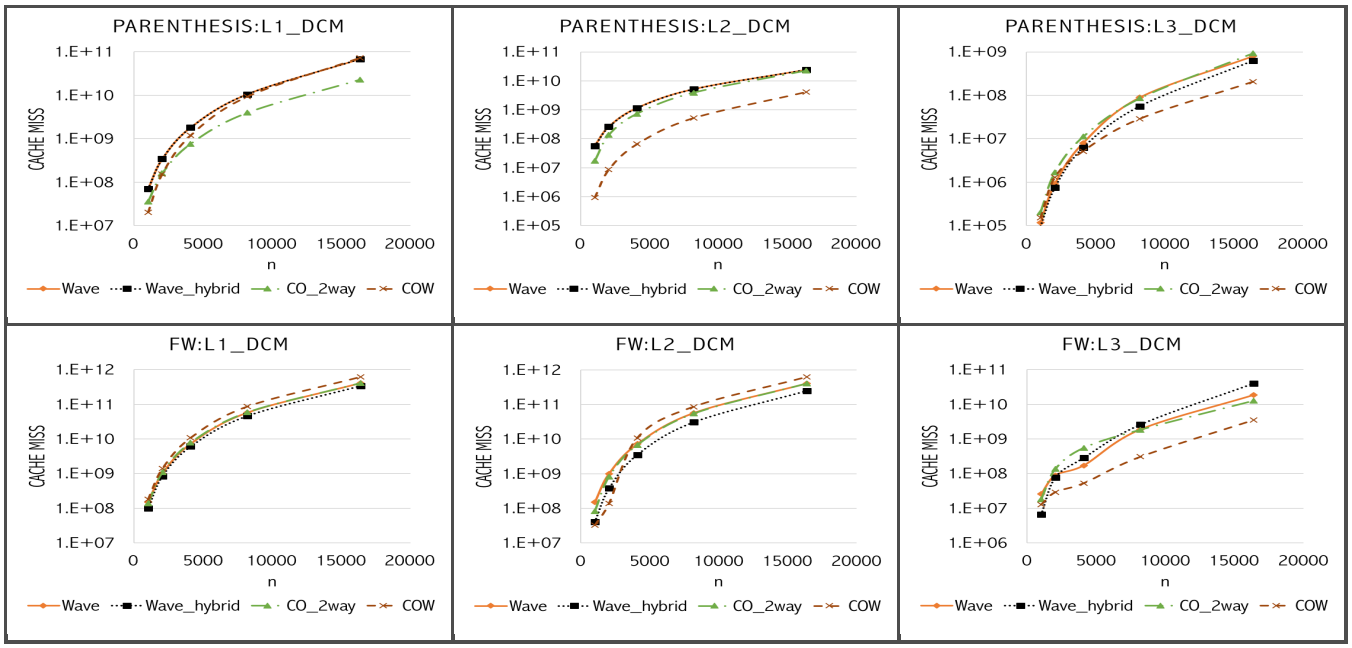

Figure 7: Cache misses in three levels of caches for classic 2-way recursive divide-and-conquer, COW and recursive wavefront algorithms for Parenthesis and 2D FW-APSP. All programs were run on 24 core machines in Comet. All implementations used cilk plus's work-stealing scheduler.

REFERENCES

[1] Comet Supercomputing Cluster. http://www.sdsc.edu/support/user_guides/comet.html. (2016).
[2] PAPI-5.3. http://icl.cs.utk.edu/papi/index.html. (2016).
[3] Stampede Supercomputing Cluster. https://www.tacc.utexas.edu/stampede/. (2016).
[4] Shaizeen Aga, Sriram Krishnamoorthy, and Satish Narayanasamy. 2015. CilkSpec: optimistic concurrency for Cilk. In *SC*. 83.
[5] Kunal Agrawal, Charles E Leiserson, and Jim Sukha. 2010. Executing task graphs using work-stealing. In *IPDPS*. 1–12.
[6] Vineet Bafna and Nathan Edwards. 2003. On de novo interpretation of tandem mass spectra for peptide identification. In *RECOMB*. 9–18.
[7] Michael A Bender, Roozbeh Ebrahimi, Jeremy T Fineman, Golnaz Ghasemiesfeh, Rob Johnson, and Samuel McCauley. 2014. Cache-adaptive algorithms. In *SODA*. 958–971.
[8] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, and Harsha Vardhan Simhadri. 2011. Scheduling irregular parallel computations on hierarchical caches. In *SPAA*. 355–366.
[9] Robert D Blumofe and Charles E Leiserson. 1999. Scheduling multithreaded computations by work stealing. *JACM* (1999), 46(5):720–748.
[10] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. *ACM SIGPLAN Notices* (2008), 43(6):101–113.
[11] Richard Brent. 1974. The parallel evaluation of general arithmetic expressions. *JACM* (1974), 21:201–206.
[12] Cary Cherng and Richard E Ladner. 2005. Cache efficient simple dynamic programming. In *AofA DMTCS*. 49:58.
[13] Rezaul Chowdhury. 2007. *Cache-efficient Algorithms and Data Structures: Theory and Experimental Evaluation.* Ph.D. Dissertation. Department of Computer Sciences, The University of Texas at Austin.
[14] Rezaul Chowdhury, Pramod Ganapathi, Jesmin Jahan Tithi, Charles Bachmeier, Bradley C. Kuszmaul, Armando Solar-Lezama Charles E. Leiserson, and Yuan Tang. 2016. AutoGen: automatic discovery of cache-oblivious parallel recursive algorithms for solving dynamic programs. In *PPoPP*.
[15] Rezaul Chowdhury and Vijaya Ramachandran. 2006. Cache-oblivious dynamic programming. In *SODA*. 591–600.
[16] Rezaul Chowdhury and Vijaya Ramachandran. 2008. Cache-efficient dynamic programming algorithms for multicores. In *SPAA*. 207–216.
[17] Rezaul Chowdhury and Vijaya Ramachandran. 2010. The cache-oblivious Gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. *Theory of Computing Systems* (2010), 47(4):878–919.
[18] Rezaul Chowdhury, Vijaya Ramachandran, Francesco Silvestri, and Brandon Blakeley. 2013. Oblivious algorithms for multicores and networks of processors. *J. Parallel and Distrib. Comput.* (2013), 73(7):911–925.
[19] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. MIT press.
[20] Alain Darte, Georges-André Silber, and Frédéric Vivien. 1997. Combining retiming and scheduling techniques for loop parallelization and loop tiling. *Parallel Processing Letters* (1997), 7(4):379–392.
[21] Jun Du, Ce Yu, Jizhou Sun, Chao Sun, Shanjiang Tang, and Yanlong Yin. 2013. EasyHPS: a multilevel hybrid parallel system for dynamic programming. In *IPDPSW*. 630–639.
[22] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. 1998. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge university press.
[23] Robert W Floyd. 1962. Algorithm 97: shortest path. *CACM* (1962), 5(6):345.
[24] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 1999. Cache-oblivious algorithms. In *40th FOCS*. 285–297.
[25] Zvi Galil and Raffaele Giancarlo. 1989. Speeding up dynamic programming with applications to molecular biology. *TCS* (1989), 64(1):107–118.
[26] Zvi Galil and Kunsoo Park. 1994. Parallel algorithms for dynamic programming recurrences with more than O (1) dependency. *JPDC* (1994), 21(2):213–222.
[27] Pramod Ganapathi. 2016. *Automatic Discovery of Efficient Divide-&-Conquer Algorithms for Dynamic Programming Problems.* Ph.D. Dissertation. Department of Computer Science, Stony Brook University.
[28] Robert Giegerich and Georg Sauthoff. 2011. Yield grammar analysis in the Bellman's GAP compiler. In *Workshop on language descriptions, tools & applications*.
[29] Georgios Goumas, Aristidis Sotiropoulos, and Nectarios Koziris. 2001. Minimizing completion time for loop tiling with computation and communication overlapping. In *IPDPS*. 10.
[30] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly-performing polyhedral optimizations on a low-level intermediate representation. *PPL* 22(04) (2012).
[31] Dan Gusfield. 1997. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press.
[32] Daniel S. Hirschberg. 1975. A linear space algorithm for computing maximal common subsequences. *CACM* (1975), 18(6):341–343.
[33] Shachar Itzhaky, Rohit Singh, Armando Solar-Lezama, Kuat Yessenov, Yongquan Lu, Charles Leiserson, and Rezaul Chowdhury. 2016. Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. In *OOPSLA*. ACM, 145–164.
[34] Kenneth E. Iverson. 1962. *A Programming Language*. Wiley.
[35] Anany V Levitin. 2009. *Introduction to Design & Analysis of Algorithms: For Anna University, 2/e.* Pearson Education India.
[36] Weiguo Liu and Bertil Schmidt. 2004. A generic parallel pattern-based system for bioinformatics. In *Euro-Par*. 989–996.
[37] Preeti Ranjan Panda, Hiroshi Nakamura, Nikil D Dutt, and Alexandru Nicolau. 1999. Augmenting loop tiling with data alignment for improved cache performance. *Computers, IEEE Transactions on* (1999), 48(2):142–149.
[38] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J Ramanujam, and P Sadayappan. 2010. Combined iterative and model-driven optimization in an automatic parallelization framework. In *SC*. 1–11.
[39] Yewen Pu, Rastislav Bodik, and Saurabh Srivastava. 2011. Synthesis of first-order dynamic programming algorithms. In *ACM SIGPLAN Notices*. 46(10):83–98.
[40] Raphael Reitzig. 2012. *Automated Parallelisation of Dynamic Programming Recursions*. Master's thesis. University of Kaiserslautern.
[41] Lakshminarayanan Renganarayanan, Daegon Kim, Michelle Mills Strout, and Sanjay Rajopadhye. 2012. Parameterized loop tiling. *TOPLAS* (2012), 34(1):3.
[42] Hiroaki Sakoe and Seibi Chiba. 1978. Dynamic programming algorithm optimization for spoken word recognition. *Transactions on Acoustics, Speech and Signal Processing* (1978), 26(1):43–49.
[43] Vivek Sarkar and Nimrod Megiddo. 2000. An analytical model for loop tiling and its solution. In *ISPASS*. 146–153.
[44] Shanjiang Tang, Ce Yu, Jizhou Sun, Bu-Sung Lee, Tao Zhang, Zhen Xu, and Huabei Wu. 2012. EasyPDP: an efficient parallel dynamic programming runtime system for computational biology. *TPDS* (2012), 23(5):862–872.
[45] Yuan Tang, Rezaul Chowdhury, Bradley C Kuszmaul, Chi-Keung Luk, and Charles E Leiserson. 2011. The pochoir stencil compiler. In *SPAA*. 117–128.
[46] Yuan Tang, Ronghui You, Haibin Kan, Jesmin Jahan Tithi, Pramod Ganapathi, and Rezaul Chowdhury. 2014. Improving parallelism of recursive stencil computations without sacrificing cache performance. In *2nd WOSC*. 1–7.
[47] Yuan Tang, Ronghui You, Haibin Kan, Jesmin Jahan Tithi, Pramod Ganapathi, and Rezaul Chowdhury. 2015. Cache-oblivious wavefront: improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency. In *PPoPP*. 205–214.
[48] Jesmin Jahan Tithi. 2015. *Engineering High-performance Parallel Algorithms with Applications to Bioinformatics.* Ph.D. Dissertation. State University of New York at Stony Brook, ProQuest Dissertations Publishing.
[49] Jesmin Jahan Tithi, Pramod Ganapathi, Aakrati Talati, Sonal Aggarwal, and Rezaul Chowdhury. 2015. High-performance energy-efficient recursive dynamic programming with matrix-multiplication-like flexible kernels. In *IPDPS*. 303–312.
[50] George Tzenakis, Angelos Papatriantafyllou, Hans Vandierendonck, Polyvios Pratikakis, and Dimitrios S Nikolopoulos. 2013. BDDT: block-level dynamic dependence analysis for task-based parallelism. In *APPT*. 17–31.
[51] Stephen Warshall. 1962. A theorem on boolean matrices. *JACM* (1962), 9(1):11–12.
[52] Michael S Waterman et al. 1995. *Introduction to Computational Biology: Maps, Sequences and Genomes.* Chapman & Hall Ltd.
[53] Michael Edward Wolf. 1992. *Improving Locality and Parallelism in Nested Loops*. Ph.D. Dissertation. Stanford University.